SWARM: A path forwards to exascale

Chris Lauderdale Rishi Khan
ET International, Inc.

1 Introduction

The requirements that will be imposed on computing hardware and software at exascale are far more restrictive than those that are imposed on present-day systems. Exascale software will need to be able to quickly express a very high degree of parallelism in order to use many CPU cores effectively; however, creating and maintaining operating-system-provided software threads makes it difficult to support a threaded execution model going forward. Furthermore, the high likelihood of hardware failure in exascale systems means that software will have to handle sudden component disappearance or configuration change. There will be a much greater degree of on-chip component heterogeneity (failure-related and in-built) in exascale systems, and optimal use of these systems will require load-balancing between these components [15].

Because maintenance of cache coherence amongst n caches requires $O(n^2)$ communication, present-day methods of mitigating memory latency will be nearly impossible, and it is likely that available memory will be broken up into non-uniform areas of varying locality to each executing core [11]; this conflicts directly with the assumption made by most present-day software that there is a quickly accessible uniform address space.

2 Overview of SWARM

SWARM (SWift Adaptive Runtime Machine) is a platform- and architecture-agnostic software runtime system that implements a new execution model in a layer between the operating system and application. It manages available hardware and software resources (*e.g.,* threads, memory, accelerators, and networking) and dynamically assigns applications' work and data to these resources as they become available, helping to maximize resource usage without needing to know *a priori* what specific resources are available. SWARM views all application-generated work in terms of ***codelets***, which are small, discrete pieces of code that can run to completion in a finite amount of time without blocking or engaging in any long-latency operations.

SWARM manages application data using an object-oriented type system that provides it with a description of type hierarchies, data layout, and construction, destruction, copying, serialization, and deserialization methods. All non-transient data used by codelets are viewed in terms of objects, to enable SWARM to automatically push and pull references, interface stubs, or entire objects between execution domains without application assistance.

Stored data and executing work are bound to particular ***locales*** within SWARM, which are related by a tree-like hierarchy with each locale possessing a scheduler and allocator to manage collective time and space, respectively, for that locale and its descendants in the hierarchy. The root (highest-level) locale refers to the entire distributed runtime and leaf (lowest-level) locales are used to manage specific threads or otherwise indivisible hardware components. The locale tree provides communications endpoints for routing application work and data, and is used by the runtime to assist with load-balancing.

3 Related work

The basis for SWARM's execution model comes from dataflow-related work by Gao et al. on the prototypical EARTH runtime [17] and later extensions to its execution model for exascale [19]. SWARM's model discards some of the theoretical limitations imposed on the earlier model and adds cancellation and continuation-passing semantics.

Other more commonly used frameworks for creating parallel and distributed software include MPI [9] and SHMEM [3] to effect multi-node distributed systems, OpenMP [6], Cilk [4], and TBB [14] for parallelizing code within a single node's homogeneous threads, and OpenCL [16], CUDA [13], and DirectCompute to enable use of heterogeneous components. Unfortunately, each of these runtimes fulfills specific roles only, and they do not tend to interoperate well (especially with regard to thread-safety). Prior work also includes ParalleX [10], which is an execution model specification for which HPX [1] exists as an implementation.

SWARM's locale hierarchy is closely related to the Hierarchical Place Trees (HPTs) used by the Habanero runtime [18], "places" in the X10 language [7], and locales in the Chapel language [5], as well as the Sequoia language's Parallel Memory Hierarchy (PMH) [8], although SWARM's locales are exposed to applications and used for collective management of both scheduling and memory management.

4 Assessment of SWARM

4.1 Challenges addressed

SWARM is designed to enable an application to rapidly expose a very **high degree of parallelism** quickly, provide a basis for dealing with **hardware failures** and **heterogeneity**, and decouple data from particular locations so that complex **memory hierarchies** can be managed.

Because application-generated work is described in terms of codelets rather than stacks and threads, SWARM can perform useful work as soon as both work and an appropriate hardware component to perform it are ready. Because no context switching or operating system interactions are needed to switch between or manage codelets, SWARM can quickly reuse threads between codelet executions, mitigating latency from long-running background operations. Codelets also offer a convenient means of dealing with hardware failures, as they offer obvious data synchronization points and minimize the amount of location-bound data. Codelets can also be used to deal well with heterogeneity; binary and data-structural differences

can be dealt with by providing multiple binary forms for codelets and using runtime-assisted object serialization/deserialization.

SWARM also offers a way to sidestep the application-level difficulties imposed by use of a deep, non-uniform memory hierarchy by managing locality and data transfer internally. Its type system enables it to reason about how data should be (de-)serialized and copied, which enables much more complete and transparent mobility of application components.

4.2 Maturity

SWARM has been applied to diverse problems such as Graph500 [12], tiled linear algebra, Barnes-Hut [2], and n-queens, and has been selected for development under the DOE XStack project (contract pending). At present, SWARM supports parallel heterogeneous programming by integrating Pthreads and runtime-managed OpenCL hooks, and can dynamically load-balance by migrating codelets amongst available CPUs and GPUs. SWARM also supports networking between individual hosts' runtimes, although data and work migration across host boundaries are still explicitly application-directed.

Overall, SWARM is still in a relatively early stage of maturity; data migration has been addressed only minimally, and further research will be needed before integrating fault tolerance or more extensive power management features. More high-level programming language development must also be done for SWARM to establish as simpler programming interface for defining codelets and describing data types.

4.3 Uniqueness and novelty

Existing runtimes and frameworks typically focus on one or two small problem areas in which to assist programs. However, since future supercomputing programs will need support for multithreading, multi-node data and work distribution, and support for heterogeneous compute elements, SWARM aims to establish a single unified framework for creation and execution of parallel programs.

The SWARM framework is DAG-based, and is concerned primarily with allowing the application to express what *can* be done at any given time, so that the runtime can determine what should be done in the present. This stands in direct contrast to most other high-performance frameworks, which typically use an imperative interface that allows an application to express what work *must* be done in the present, while the application waits for the work to complete. Even other DAG-based runtimes typically require dependencies between application components to be declared at compile time or program startup, which works well only for a fairly small class of programs.

Although other frameworks such as Cilk or OpenMP can handle distinct forms of multithreaded parallelization (typically fork/join-style since it meshes well with existing execution models), SWARM supports recursive, bulk-synchronous, producer-consumer, streaming, and graph-traversal programs well; it reduces all parallelization to primitives that notify the runtime of work being available, then allows higher-level constructs to be used on top of those primitives. Other frameworks typically build parallelization around primitives with specific purposes that restrict the kinds of problem class for which the frameworks are appropriate.

4.4 Applicability

One of the design goals of SWARM was to establish a unified computing model as the basis for many different application types, not just exascale programs. Traditionally, fork/join-based models have been most popular since they can be overlaid on software threads without much work; Cilk and OpenMP are two frameworks that provide support for recursion-based and bulk-parallel fork/join models, respectively. However, these models are less appropriate for imbalanced or flow-based workloads, such as many physical simulations, graph algorithms, or streaming applications, which typically use a special-purpose DAG-like model to manage workflow. SWARM provides a model that can be used to implement all of the above schemes in a scalable, low-overhead fashion.

SWARM is designed to be platform- and architecture-agnostic, so it is portable to many different kinds of computing system from embedded platforms to supercomputers. This allows applications and application components to be written and reused in vastly different situations, and allows programs to be scaled up or down dramatically, or distributed widely or locally, without loss of functionality or need for modification.

4.5 Usability

The primary difficulties in using SWARM stem from its execution model and type system. Because SWARM and SWARM software are currently programmed in C, each codelet fork must be established as a separate C function, referenced indirectly by the codelet's data structure in memory. Breaking up program flow like this is unnatural for most programmers, and can be difficult to master. SWARM's type system is applied to C data structures, and must rely on hand-coded metadata structures to enable SWARM to inspect or operate upon objects. Casting and coercing between object-oriented types must also be done by hand, since C itself won't adjust pointers correctly. These problems are largely cosmetic, and to circumvent them, ETI is developing SCALE (SWARM Codelet Association Language Extensions), which constitutes a set of language extensions to C that simplify creation of, and interaction with, codelets and SWARM's type system.

5 Conclusion

Reaching exascale cannot simply be a matter of continuing present-day practices and expecting them to work at massively different scales. In order to overcome the problems that are currently being experienced and that will be experienced as the exascale is approached, SWARM establishes a new execution model and software runtime layer to help software express a high degree of parallelism, simplify use of heterogeneous components, manage data and work migration, and provide a basis for hardware fault tolerance.

References

[1] M. Anderson, M. Brodowicz, H. Kaiser, and T. Sterling. An application-driven analysis of the ParraleX execution model. Technical report, Louisiana State University, Baton Rouge, LA, USA, Sep. 2011.

[2] J. Barnes and P. Hut. A hierarchical $O(n \lg n)$ force-calculation algorithm. In *Nature*, 324(6096):446–449, Dec. 1986.

[3] R. Barriuso and A. Knies. *SHMEM user's guide for C*. Cray Research, Inc., Eagan, MN, USA, June 1994.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30:207–216, August 1995.

[5] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *IJHPCA*, 21(3):291–312, 2007.

[6] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable shared-memory parallel programming*. The MIT Press, 2007.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA '05*, pages 519–538, New York, NY, USA, 2005. ACM.

[8] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[9] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 2nd edition, Nov. 1999

[10] H. Kaiser, M. Brodowicz, and T. Sterling. ParalleX: An advanced parallel execution model for scaling-impaired applications. *ICPPW*, pages 394–401, Sep. 2009.

[11] P. Machanick. Approaches to addressing the memory wall. Technical report, University of Brisbane, Brisbane, QLD, Australia, 2002.

[12] R. C. Murphy, K. B. Wheeler, B. W. Barrett, J. A. Ang. Introducing the Graph 500. Cray User's Group (CUG), May 5, 2010.

[13] NVIDIA Corporation, Santa Clara, CA. *NVIDIA CUDA programming guide*, June 2007.

[14] J. Reinders. *Intel Threading Building Blocks*. O'Reilly Media, Sebastopol, CA, July 2007.

[15] J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. In *VECPAR 2010*, volume 6449 of *Lecture Notes in Computer Science*, pages 1–25. Springer Berlin/Heidelberg, 2011.

[16] J. E. Stone, D. Gohara, G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–72, May 2010.

[17] K. B. Theobald. *EARTH: An efficient architecture for running threads*. PhD thesis, McGill University, Montreal, Que., Canada, 1999.

[18] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. Hierarchical Place Trees: A portable abtraction for task parallelism and data movement. In *Proceedings of the 22nd Workshop on Languages and Compilers for Parallel Computing*, Oct. 2009.

[19] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Position paper: Using a "codelet" program execution model for exascale machines. In *EXADAPT '11*, New York, NY, USA, June 2011. CAPSL, ACM.